TerraWeather
Ensuring Your Safety & Success

AI Workshop – Lecture 9

Data Transformations

9.0 Neural Networks are not black boxes

From the previous lectures, you should clearly see that neural networks are not “black boxes” into which you can dump data into, expecting your answers to come out as output!

Specifically:

- Using your raw input data “as is” rarely results in a successful network with good generalization. This is because the time-aspect of the data is not captured.

- Naïve use of a sliding window of raw data to capture the time aspect of the data also does not give good results, because (1) the inputs within a window are highly correlated (2) the “meaning” of each input will change as the window is slid forward in time.

The answer is to *transform* the input over a given window, creating a very high-dimensional input that acts like a “fingerprint” that signals changes in the time-series. This way, you can give your NNs much more data to work with, to find patterns. Figure 9.1 illustrates this process:

Figure 9.1 – Transforming raw data over a window.

9.1 The Need for Data Transformations

Why is this necessary? The answer is that neural networks are essentially feature detectors. They can only work to detect features already present in the data stream. They can't create features from nothing.

For example, consider the following input data (red, underlined) and label(blue) training pairs:

(1,2,3; **6**) , (4,1,6; **11**) , (5,10,34; **49**) , (23,17,18; **58**) ...

Can you discern the pattern in the inputs? The labels are just the sum of the

inputs. Eg, 49 = 5+10+34. This may seem to be a trivial transformation (summing) but neural networks are not able to transform the data themselves. They need you to do it for them. They can only discern patterns in the data, if they are present, not come up with transformations that may yield patterns.

The data transformation works by creating these features – they create features that are difficult or impossible for NNs to do by themselves.

Note that with this method:

1. With the right transformations, the inputs should become highly decorrelated,

2. The “meaning” of each input corresponds to a single type of transformation, so they remain the same, even as the window is slid forward.

We'll discuss some useful transformations next.

9.1 Thresholding

Many processes have natural *critical thresholds* (eg, a machine's operating envelope, the right temperature for mosquito breeding, the freezing temperature of water, etc.) which determine the absence/presence of a phenomenon of interest. These thresholds should be apparent from a knowledge of the problem domain.

The thresholding transformation might be as simple as:

$$f(x;x_0)=\begin{cases}0, x<x_0\\1, x\geq x_0\end{cases}$$

A better version might indicate the distance from the threshold, using a sigmoid or tanh function. (Quiz 9A.1). Thresholding essentially incorporates domain knowledge directly into your inputs.

9.2 Moments

The *k-th centred moment* of a data window $\{x_1 \ldots x_N\}$ is approximated by:

$$m(x;k) = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^k$$

where μ is the average value of $\{x_1 \ldots x_N\}$. The first moment is always 0, the second moment is the variance, while the third and fourth moments are known as the *skewness* and *kurtosis* respectively.

Moments can be useful in smoothing noisy time-series data. In Autocaffe, there is a **`moments`** prefab.

9.3 Discrete Fourier Transform (DFT)

The Discrete Fourier Transform is an often used tool for analyzing periodic time-series. This transformation essentially breaks down the time-series as a linear combination of sine waves with increasing periods. There are many resources describing DFT, so we will not cover this in detail here.

DFTs are useful because they are able to pick out strong periodic signals in your input. Needless to say, this transform is most useful for periodic input (eg, load forecasting).

There are many software packages to compute the DFT, often using the “Fast Fourier Transform” technique. There is one available in **`numpy`**.

To use it, you have to preprocess your input signal using DFT (with numpy, for example), then select the DFT output (called the *power*) at frequencies you are interested in as inputs. The natural frequencies of the problem domain are often good choices. These will appear as consistent “spikes” in the DFT power spectrum.

Note that you should perform the DFT over windowed data, so as to capture any changes in these spikes as the window moves in time.

9.4 Momentum and Force

Another transformation that is useful is to calculate momentum and force:

$$\begin{aligned} M_k &= x_k - x_{k-1} \\ F_k &= x_k - 2x_{k-1} + x_{k-2} \end{aligned}$$

These transformations only makes sense if the underlying data comes from a over data from a sliding window in time.

9.5 Combining Transformations

You can combine these transformations to obtain more complex ones. For example, you can run a threshold on momentum, or calculate the moments then get the force from this datastream. The possibilities are endless.

Quiz 9A

- Create a better thresholding function using a sigmoid or hyperbolic tangent. This transformation should be continuous. What parameters does your transformation have?

TerraWeather
Ensuring Your Safety & Success

Lab 9: Thresholding and Moments

In this lab, you will
1. Use the moments layer to transform input
2. Add momentum and force loss using prefabs

Instructions

Moment layer
The moment layer calculates the mean and higher central moments for a moving window of adjustable size and has the following parameters

- **window_size:** sets the size of the window (number of time steps) over which to calculate the moments.
- **axis**, **split** and **once**: These are similar to the norm layer, with axis and split determining which axis of the blob onward will be lumped together for the moment calculation, and the once parameter providing the option for the moment calculations to be done only on the first iteration.
- **moments**: select the order of moment to be calculated. If set to 1, the mean is returned, else the k^{th} moment is calculated using the equation

$$m = \frac{1}{N} \sum_{i=1}^{N} \left(x_i - \bar{x}\right)^k$$

where N is the window size and x the mean.

Moments are not calculated for the first (window_size – 1) time steps, since there is not enough time steps to fill the window. These time-steps are thus filled with zeros and should be
The format for using the moment layer prefab is

```
<layer> <window_size> <moment_order> <axis> <split?> <once?> moment
```

Mini: Words and local variables
We introduce here 2 features of Mini that helps simplify your network definition files.

Words

Words are groups of commands given a name. The group of commands can be executed by calling the word name. This is useful for grouping commands that together perform a certain task, or for often repeated code segments.

Words are defined by wrapping a group of commands between a colon and semi-colon like so

```
: <word_name>
     command1
     command2
     ...
;
```

Notice that the network definition itself is a word called network. We will define other words to keep the network word simple and uncluttered.

Local variables
Local variables gives a name to the object on the stack. This name is valid only within the word that it is defined in. Local variables are used to save certain layers so that they can be recalled and used as input to other layers later on in the network definition.

To declare a local variable, use the curly brackets with a name in between

```
{ local_variable_name }
```

This will bind the word `local_variable_name` to the top object on the stack. Note the space between the brackets and the variable name. The prefab layer commands automatically puts the created layers onto the stack, so to save a particular layer, declare the local variable immediately after the layer command. Example:

```
"train" TrainSize true hdf5data train [tops] data
{ data }
```

The local variable declaration has to be done on the next line in this example, as otherwise the `[tops]` word will read it as top blob names of the `hdf5data` layer.

Network Structure
We define 2 words to handle the data loading and transformation.

Load
The **`load`** word consists of the input layers and normalisation of both data and label blobs. This word does not have any input and outputs a normalised data blob and label blob on the stack, which is denoted by the comment `( -- data label )`.

```
: load ( -- data label )
       "train" TrainSize true hdf5data train [tops] data label
       "test" TestSize true hdf5data test [tops] data label
       '$data 1 true TrainSize true norm
       1 723 truncated-train/test drop
       '$label 1 false TrainSize true norm
       1 723 truncated-train/test drop
;
```

The `'$data` and `'$label` words represent the `data` and `label` blobs output from the hdf5data layers above.

Transform

The transform word takes in a data layer/blob and outputs the transformed blob on the stack.

```
: transform ( data -- layer )
       1 1 bisect [tops] ? ?
       dup { dengue }
       '$2 { data }
       {{ dengue
          data 2 2 1 select
          data WindowSize 1 1 true true moment
          data WindowSize 2 1 true true moment
          data WindowSize 3 1 true true moment
          data WindowSize 4 1 true true moment
       }} 1 concat-seq
       WindowSize truncate
;
```

Line 1: this bisects the blob to extract the dengue data, which is at axis 1, index 0. The two unnamed tops '`?`' will be given automatically generated names.

Line 2: `dup` duplicates the bisected blob on the stack. One copy of the bisected blob is then consumed by declaring it as the local variable `dengue`.

Line 3: `'$2` takes the remaining copy of the bisected blob and selects the second top blob (ie the blob containing axis 1, index 1 and 2 of the original data blob, which has the temperature and rainfall data). This blob is given the variable name `data`.

Lines 4-10: The lines within the {{ and }} brackets form a sequence and the transformations that are included in the output blob. The first two lines give the dengue and temperature data respectively. This is followed by the first 4

orders of moment applied on the `data` blob (containing temperature and rainfall data). All these transformations are then concatenated together along axis 1 using the `concat-seq` word

concat-seq	`<seq> <axis> concat-seq`
	Concatenates all the layers/blobs in sequence `<seq>` along axis `<axis>`

Line 11: This line truncates the first `WindowSize` time steps to remove the zeroed data done by the moments layer.

The transformed data will then be fed into the inner product layers to produce a prediction.

Network
The main network word in `train.m` is given below

```
: network
       load { label }
       transform

       named Dengue
              NumLayers NetworkSize 1 active mlp
              1 1 innerproduct activate
       end-named

       label WindowSize truncate
       16 20 1 select
       loss [tops] loss
;
```

The last 3 lines truncate the labels to match the dimensions of the prediction blob and selects the T+16 label values, before comparing it with the predictions in a loss layer.

To complete the `train.m` file, we need to include the prefab libraries and define the constants used in the network definition. These lines should be placed at the top of the file.

```
uses layers learning

${ $ SIZE . }       => NetworkSize
${ $ LAYERS . }     => NumLayers
${ $ WINDOW-SIZE . }     => WindowSize
729    => TrainSize
```

```
911    => TestSize
```

test.m

The test network definition is almost the same as the training network. The only difference is the removal of the `truncated-train/test` lines in the `load` word, the lack of a loss layer and the relabeling of the prediction and label blobs, shown in bold below.

```
uses layers learning

${ $ SIZE . }        => NetworkSize
${ $ LAYERS . }      => NumLayers
${ $ WINDOW-SIZE . }     => WindowSize
729    => TrainSize
911    => TestSize

: load ( -- data label )
       "train" TrainSize true hdf5data train [tops] data label
       "test" TestSize true hdf5data test [tops] data label
       '$data 1 true TrainSize true norm
       '$label 1 false TrainSize true norm
;

: transform
       \ same as train.m
;

: network
       load { label }
       transform

       named Dengue
              NumLayers NetworkSize 1 active mlp
              1 1 innerproduct activate
       end-named
       relabeled prediction_16

       label WindowSize truncate
       16 20 1 select
       relabeled label_16
;
```

config.txt

The expander variables `SIZE, LAYERS` and `WINDOW-SIZE` are defined in `config.txt`, given below. The number of training iterations is also added as a variable to allow it to be easily changed.

```
SIZE : #{ 8 16 2 list as SIZE } ${ $ SIZE . }
```

```
LAYERS: #{ 1 2 2 list as LAYERS } ${ $ LAYERS . }
WINDOW-SIZE: #{ 20 1 list as WINDOW-SIZE } ${ $ WINDOW-SIZE . }

TRAIN_ITERS: #{ 10000 1 list as ITERS } ${ $ ITERS . }
TEST_ITERS: 1

DATASET: dengue-2
REPEAT: 2
```

solver.prototxt
The test and display interval are set to trigger 200 times over the training phase, while the step size being a tenth of the total iterations will reduce the learning rate 10 times from 0.001 to 0.00001.

```
net: "train.prototxt"
type: "SGD"
test_iter: 1
test_interval: ${ $ ITERS 200 / . }
base_lr: 0.001
lr_policy: "step"
stepsize: ${ $ ITERS 10 / . }# drop lr every n iterations - 1/10
gamma: 0.599484  # lr from 0.001 to 0.00001
display: ${ $ ITERS 200 / . }
max_iter: ${ $ ITERS . }
snapshot: ${ $ ITERS . }
snapshot_prefix: "lab9_${ $ counter (.) }"
solver_mode: CPU
```

report.tex_template
Copy this from a previous lab. Use these lines immediately after the first line to display the configuration in the report.

```
\textbf{Layer configuration} \\
MLP Size: ${ $ SIZE . } \\
Layers: ${ $ LAYERS . } \\
Window Size: ${ $ WINDOW-SIZE . }
```

metrics: Remember to put a copy of this file into your experiments folder.

Explore
Train the network using the repeat task. Experiment with different configurations using the expander variables. You can also add more variables to expand to explore different parts of the configuration space.

Try to implement the above network with momentum and force loss using

prefabs. You can use the layers below

momentum*	`<t-1> <t> momentum*`
	Finds the momentum by subtracting the t-1 blob from the t blob
force*	`<t-2> <t-1> <t> force*`
	Finds the force from the 3 input blobs

Homework
The data transformation covered in this lecture is applied on an earthquake prediction problem in this paper:

http://onlinelibrary.wiley.com/doi/10.1002/2017GL074677/abstract

We highly recommend you to read this paper for an example of a successful application of these techniques.